A DESIGN OF OPERATING SYSTEM FOR EASILY IMPLEMENTING EFFICIENT FILE SYSTEMS

性能のよいファイルシステムの実装を容易にするＯＳの設計

by

Kenichi Kourai

光来健一

A Senior Thesis

卒業論文

Submitted to

the Department of Information Science

on February 18, 1997

in partial fulfillment of the requirements

for the Degree of Bachelor of Science

Thesis Supervisor: Takashi Masuda　益田隆司

Title: Professor for Information Science

ABSTRACT

The OS kernel should provide a framework with which programmers can easily implement an efficient file system. With conventional frameworks, a file system has been embedded into the kernel for efficiency or otherwise constructed at the user level for ease of implementation. However, the former technique makes the implementation difficult, and the latter involves performance penalties. This thesis proposes a framework with that programmers can first implement a file system at the user level and then embed it into the kernel for the final release. To do this without changing the source code of the file system, this framework provides an abstract interface to hide differences between the kernel level and the user level. To show the feasibility of our idea, we extended the kernel of NetBSD, implemented the proposed framework, and actually constructed file systems on top of it.

論 文 要 旨

ＯＳカーネルは、性能のよいファイルシステムを、容易に実装できるようにする枠組を提供するべきである。従来の枠組では、性能をよくするためにカーネルの中に作り込む方法か、実装しやすいようにユーザプロセスとして作成する方法が主流であった。しかし前者は実装がしにくいという点が問題であり、後者は性能が悪くなるという点が問題である。本研究で提案する枠組では、ファイルシステムをユーザレベルで実装し、完成後、カーネルの中に組み込むことができる。ユーザレベルで実装したファイルシステムをそのままカーネルに組み込むことができるように、この枠組は、カーネルとユーザレベルの違いを隠蔽する抽象インタフェースを提供する。さらに本研究では、NetBSD のカーネルを拡張し提案した枠組を実装し、その上で実際のファイルシステムを構築する。

Acknowledgements

I would like to thank my supervisor, Prof. Takashi Masuda, for his significant advice and generous encouragement. I am also thankful to Dr. Shigeru Chiba for his valuable suggestions and useful discussions. I would be grateful to all the members of the Masuda laboratory for many suggestions and their encouragement.

Table of Contents

Acknowledgements **i**

1. Introduction **1**

2. Background **4**

2.1 Needs of the ability to implement a new file system 4

2.2 Conventional approaches 5

2.2.1 Monolithic kernel 6

2.2.2 Microkernel 7

2.2.3 Extensible kernel 9

2.2.4 Summary 11

3. Two-phase implementation **12**

3.1 What is two-phase implementation? 12

3.1.1 Overview 13

3.1.2 Mechanisms of two-phase implementation 14

3.1.3 Protocol between the kernel and a user level file system . . . 16

3.1.4 Passing data between the kernel and a user level file system 21

3.1.5 Data structure used in the communication 22

3.2 Extension of the kernel 23

3.2.1 Overview 23
3.2.2 Implementation of upcall 24
3.2.3 Kernel protection 25
3.2.4 System calls for a user level file system 26
3.3 User level library 27
3.3.1 Overview 27
3.3.2 Multiplexing of the file system 28
3.3.3 Higher-level abstraction 29
3.4 Kernel level library 30

4. Experiment **32**

5. Conclusion **36**
5.1 Summary 36
5.2 Future work 37

References **38**

A. A program for our experiment **41**

List of Figures

2.1 The communication between the Mach kernel and the file server in Mach-US. 9

3.1 The concept of the debug phase 14

3.2 The concept of the release phase 16

3.3 The protocol 1 between the kernel and a user level file system . . . 17

3.4 The protocol 2 between the kernel and a user level file system . . . 19

3.5 The protocol 3 between the kernel and a user level file system . . . 20

List of Tables

2.1 The comparison between three kinds of conventional kernels. 11

4.1 The number of upcalls to the user level SMFS. 33

4.2 The time spent on copy of 64k byte and the ratio to the original SMFS. 34

Chapter 1

Introduction

Since a file system is a major performance factor of operating systems, the file system should be customized to fit the applications, particularly, for mobile computing and multimedia. To do this, a file system needs to be not only efficient but also easy to implement.

Previous operating systems have taken different approaches for this issues. The conventional monolithic kernel pursues the efficiency of a file system. Therefore, the file system is embedded into the kernel and is executed in the kernel space so that the file operations are very efficient. However, this approach makes it difficult to implement. For example, we must debug the file system in the kernel. The microkernel, on the other hand, pursues ease of implementation of a file system. Therefore, the file system is implemented as a server in the user space. Because of this, the modularity of the operating system is improved and implementing a file system is made easy. However, the problem of this approach is inefficiency since the kernel and the file server need to communicate frequently.

To solve this dilemma, we propose a new approach, called two-phase implementation. Two-phase implementation divides a development process of a file system into a debug phase and a release phase so that an efficient file system can be

easily implemented. In the debug phase, we pursue the ease of implementation of a file system as in the microkernel, so the file system is implemented as a user process and debugged in the user space. In the release phase, we pursue efficiency of the file system as in the monolithic kernel, so the file system that we finish to debug is embedded into the kernel without changing source code. Thus, two-phase implementation satisfies two antithetic properties: ease of implementation and efficiency.

To make it possible to embed the file system in the kernel without changing the source code, we provide the abstract interface to absorb differences between the kernel level and the user level. Additionally, the programmers are released from writing complicated codes because the abstraction also encapsulate low level data structures in the kernel.

The operating system that enables two-phase implementation consists of three components. One is the extended mechanism of the kernel, which performs upcalls toward a user level file system, in the debug phase. Another is the user level library that handles the upcalls from the kernel and also translates the kernel-level data structures to the user-level structures represented with a higher-level abstraction, in the debug phase. The other is the kernel-level library to enable the file system implemented at the user level to embedded into the kernel without changing source code, in the release phase.

We implemented this system on NetBSD 1.2 running on SPARCstation 5. Also, we measured the performance of a simple file system implemented on top of this system so that the file system is nearly as efficient as the equivalent system directly implemented in the kernel without our mechanism.

The rest of this thesis is organized as follows. First, Chapter 2 describes the conventional approaches and the issues. Then, Chapter 3 proposes our idea of two-

phase implementation and presents details of the idea. Chapter 4 measures the performance of a file system implemented with two-phase implementation. Finally, Chapter 5 concludes this thesis.

Chapter 2

Background

In this chapter, we describe why the kernel needs the ability to implement a new file system and then the approaches of the conventional systems and the issues.

2.1 Needs of the ability to implement a new file system

As the technology of mobile computing and multimedia is advancing, the file systems that are customized to fit the applications are required. For mobile computing, for instance, the file system which can keep consistency and which takes account of the difference of the circuit speed is desirable. For multimedia applications, the file system which can transmit the large amount of the data efficiently or which can change the way of read/write according to the contents of the data is desirable.

A file system influences the system performance largely. Even if only CPU and memory are fast, the whole system performance does not increase if the file system is inefficient. Needless to say, the file system for disk whose access speed is extremely slower than the memory access speed must be efficient. Moreover, an efficient file system for memory is also required because the opportunity where we

use memory as the RAM disk increases as the price of memory falls.

However, we also desire that it is easy to implement file systems. If it is difficult to implement them, it takes more time to implement new file systems which researchers design. This has a negate impact on the advance of file systems. Of course, ease of implementation, which we say here, means not only to write file systems, but also to debug them.

Efficiency and ease of implementation are, however, the antithetic concepts each other. As we try to create more efficient file system, it becomes more difficult to implement. Conversely, as we try to be able to implement file systems more easily, it becomes more inefficient. This is similar to the relation between an assembly language and a higher-level language. If we write programs in an assembly language, we can write everything that we want and the programs are fast, but it is difficult to write them. On the other hand, if we write programs in a higher-level language, we may not be able to write everything that we want, but we can write the programs more easily.

To solve this dilemma and promote the research of file systems more, the OS kernel should provide a framework with which programmers can easily implement an efficient file system.

2.2 Conventional approaches

The conventional approaches are classified into three categories, roughly speaking. One is the approach of the monolithic kernel such as UNIX [17]. Another is the approach of the microkernel such as Mach [1]. The other is the approach of the extensible kernel such as SPIN [3, 4]. In this section, we describe the feature of these conventional approaches, and then their disadvantages.

2.2.1 Monolithic kernel

In the monolithic kernel, all facilities that the operation system should possess are in the kernel. Of course, all file systems are also embedded into the kernel. Therefore, the overhead to switch the context for the file systems is nothing, and we can get maximum efficiency. However, we must implement a new file system in the kernel directly if we want to implement it. Although the current monolithic kernel is modularize, and we can use dynamically loadable modules, it is difficult that we implement a new file system in the kernel because the modules run in the kernel space after all. The reasons of the difficulty are as follows:

- Difficulty of the debug
- Crash of the kernel by some bugs
- Much time for linking
- Requirement of the detailed knowledge about the kernel

First, the usage for kernel debuggers is restricted, although we can use them. There are gdb and ddb for the kernel debuggers used in NetBSD 1.2. Gdb is used for analizing the core file which the kernel dumps in the crash. We can not always analize the core file the kernel because the environment where the kernel dumped the core file and that where we try to analize it are different. Additionally, it takes much time to dump the core file. Ddb can, on the other hand, debug the kernel at the runtime. However, it is very difficult to debug the kernel with ddb, (1) because ddb runs on PROM monitor and we need another machine to see the kernel source, and (2) because the runtime kernel does not include any debug information and therefore we must debug the kernel in the assembler level.

Second, we must reboot the whole system when the kernel crashed for some bugs of file systems. It takes extremely longer time than any other operations because the kernel must check all file systems using fsck which had been mounted in the crash. This causes the programmers that implement a file system to get nervous. Furthermore, by the crash, some file systems may be broken so badly that it is impossible to fix them.

Third, it takes much time to link all object files for the kernel because the monolithic kernel tends to become bigger and bigger. Because of this, time for the link of the kernel does not change, even if the code of the file system that programmers write is very short. Recently, however, this problem is solved by modularizing file systems, and not linking to the kernel statically.

The fourth may not be a problem for the experts of the operating system, which know the internal structure of the operating system fully. But this is also one of problems for us because our primary aim is to make more people possible to implement file systems. To write a file system in the kernel, we must understand many complicated structures used in a file system and enormous related functions. Therefore, the experts may prefer ease of implementation for writing a file system in the kernel.

2.2.2 Microkernel

The microkernel is smaller than the monolithic kernel because many facilities are realized out of the kernel as one OS server or some servers. Likewise, file systems are also exported out of the kernel, and are executed in the user space. Therefore, the two of four factors in 2.2.1 to make it difficult to implement file systems, that it is difficult to use the debugger and that the kernel crashes by bugs, are eliminated. So it is much easier than in the monolithic kernel to implement file systems.

First, we can freely use the debugger to debug the file system on the microkernel because the file system is running at the user level like other application programs. This makes programmers to shorten the period to develop a new file system.

Second, some bugs in the file system do not cause the kernel to crash. Because the kernel and the file system is detached in the kernel space and the user space, respectively, the kernel is safe even if the file system crashes. When the file system crashed, we only restart the file system again. In short, we need not wait for the reboot of the kernel.

By the way, if the system on top of the microkernel is a single server system, the time for the link of a file system is almost same as that in the monolithic kernel since one OS server is as huge as the monolithic kernel. If the system is a multiserver system, which has some cooperative servers, however, the time for the link is decreased extremely because a file system is compiled and linked as one file server.

As you can see above, the microkernel makes it much easier to implement a file system, but a new problem arises. It is that the efficiency of a file system is sacrificed. Let us consider the case of Mach-US [8], which is the multiserver system on top of Mach. Figure 2.1 shows the communication between the Mach kernel and the file server of Mach-US. The communication is done as follows.

1. The system call for files, which is issued by the application program, is sent to the file server through the emulation library linked to the application program by means of Mach IPC [2].

2. The file server executes the system call.

3. If the file server requires the extension of the kernel facility such as device I/O, it calls the Mach kernel.

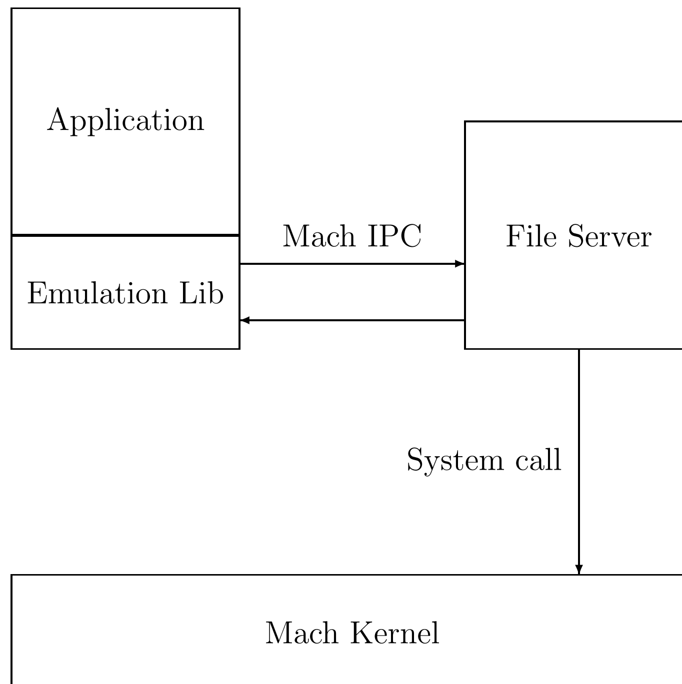

Figure 2.1: The communication between the Mach kernel and the file server in Mach-US.

If the file server does not call the Mach kernel, the context switch happens only two times as well as that of the system call for files in the monolithic kernel. However, if the file server calls the Mach kernel, the context switch happens four times even at the minimum.

2.2.3 Extensible kernel

SPIN, which is one of extensible kernels, can describe a file system as an extension. The extension is written in Modula-3 [14], which is a general purpose programming language with interfaces, objects and type safety. For these features, it is safe that the kernel executes the extension written in Modula-3. The extension is linked to the kernel dynamically after the kernel boots, and executed in the kernel space.

This approach is similar to the idea of a dynamically loadable module in the monolithic kernel, with which we can add functionality to the kernel dynamically. But there are at least two differences. One is the flexibility. In the dynamically loadable module, the only superuser can load the modules into the kernel; in SPIN, arbitrary users can link the extensions into the kernel. The other is the safety. The dynamically loadable module may be dangerous because it is loaded into the kernel without any safety checks, whereas the extension of SPIN is safe because it is written in type-safe language, Modula-3, and is checked also at runtime.

However, SPIN has two disadvantages. First, the extension of SPIN must be written in Modula-3. In general, the code of the operating systems is written in C, and moreover Modula-3 is not popular to many programmers. In Modula-3, we may not be able to write some codes that we can write in C because Modula-3 is more restricted than C. Second, the servers of SPIN are efficient if we implement most parts of the servers as extensions, but it is difficult to debug the extensions written in Modula-3. On the other hand, it is easy to debug the servers if we implement most parts of the servers in the user space without extensions, but the servers are inefficient. In short, SPIN can not satisfy both ease of debug and efficiency at the same time.

2.2.4 Summary

In this section, we described the approaches of three kinds of conventional systems. We summarize the features of their approaches in Table 2.1.

	Abstraction	Debuggability	Safety of the kernel	Efficiency
UNIX (Monolithic kernel)	low	low	low	high
Mach (Microkernel)	low	high	high	low
SPIN (Extensible kernel)	high	variable	variable	variable

Table 2.1: The comparison between three kinds of conventional kernels in abstraction, debuggability, safety of the kernel and efficiency. The debuggability, safety of the kernel, and efficiency of SPIN change largely, depending on the amount of extensions. If the extensions are little, the debuggability and safety of the kernel are high, but the efficiency is low. Conversely, if the extensions are large, the efficiency is high, but the debuggability and safety of the kernel are low.

Chapter 3

Two-phase implementation

In this chapter, we explain two-phase implementation that we propose, and describe the details of the implementation of (1) the kernel extension, (2) the user level library for supporting a user level file system, and (3) the kernel level library for supporting the file system embedded into the kernel.

We have used NetBSD 1.2 on SPARCstation 5. However, because we have written only machine independent code, you will be able to run this system on any other machines which NetBSD 1.2 supports.

3.1 What is two-phase implementation?

We propose a new approach, which possesses better properties of the monolithic kernel and of the microkernel. We call this approach two-phase implementation. In this section, we first describe the overview of two-phase implementation, and then explain the mechanism. Next, we discuss the issues for realizing two-phase implementation.

3.1.1 Overview

Two-phase implementation is to implement a file system at the user level and then to embed it into the kernel after the debug. We call the former a debug phase; the latter, a release phase.

In the debug phase, the file system implemented on our system runs in the user space. Because the file system runs in the user space as one regular process, we can use debuggers with which we can debug the file system at runtime and at source level. Therefore, the time to debug a file system shortens, although the file system is inefficient.

In the release phase, the file system that we created runs in the kernel space because we embed it into the kernel after we finish to debug it. Therefore, the efficiency of the file system is improved dramatically, comparing with that in the debug phase. As a result, the file system is executed as efficiently as what we implement in the kernel from the beginning.

Thus, two-phase implementation, separating the implementation of a file system to two phases, can satisfy two antithetic properties: ease of implementation and efficiency.

In addition to the above primary feature, our approach has one more feature. We can use a higher-level abstraction to expose kernel data, so our system does not only support to debug file systems, but also makes it easier to implement them because of the abstraction. Although the abstraction somewhat restricts the ability of the file system written with it, that is a small problem because our aim is to make it easier to implement file systems.

Of course, due to the abstraction needed to hide differences between the kernel level and the user level, the efficiency of the file system is decreased a little, but

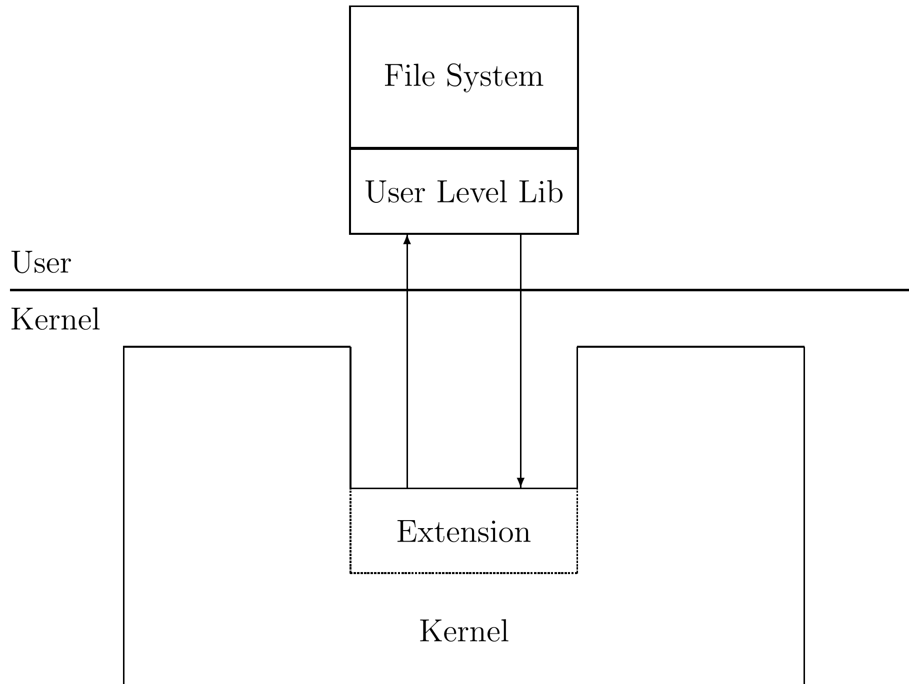

Figure 3.1: The concept of the debug phase

this degradation is not significant generally, although it depends on the degree of the abstraction.

3.1.2 Mechanisms of two-phase implementation

We need two different mechanisms for two-phase implementation because the debug phase pursues ease of implementation, and the release phase pursues efficiency.

In the debug phase, the mechanism with which a user level file system can process file operations is required. Figure 3.1 shows the concept of the system. In this figure, the file system exists in the user space. This system runs as follows.

1. The kernel is called by the system call of an application program.

2. In the system call, if it is necessary to execute the file operation for the user

level file system, the kernel does an upcall toward the user level file system by means of a signal. The file operation means the operation which the kernel is designed so that it dispatches according to file systems, such as `VFS_MOUNT`. At this point, the process that causes the upcall sleeps.

3. When the library which is the base of the user level file system receives the upcall from the kernel, the library issues a system call, `ufgetop`, in order to get the kind of file operation and the data needed for the file operation.

4. According to the file operation, the library calls the function that programmers wrote in the file system and that executes the file operation. At this time, the kernel level data structures are translated to the user level data structures with a higher-level abstraction.

5. If this function requires to execute the functions in the kernel, the system call to do so is issued through the library.

6. The results of the file operation are returned to the kernel, and finally the process sleeping in the kernel is waken up.

7. The first system call from the application program repeats such upcalls zero or more times, and then is finished.

In the release phase, the user level file system implemented on our system is embedded into the kernel, so the environment is same as other file systems which are implemented in the kernel from the beginning, except some differences such as that of the abstraction. To hide the differences, we use a kernel level library. Figure 3.2 shows this concept. This system runs as follows.

1. The kernel is called by the system call of an application program.

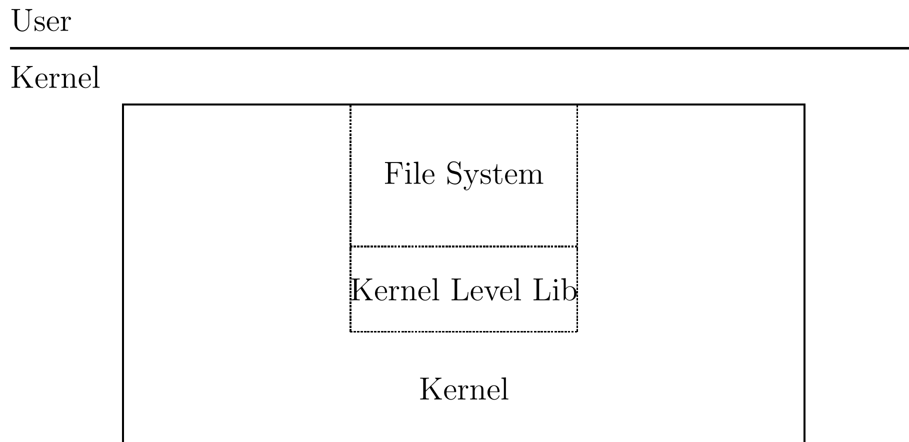

Figure 3.2: The concept of the release phase

2. In the system call, if it is necessary to execute the file operation for the file system newly embedded into the kernel, the kernel calls the kernel level library.

3. The library translates the kernel level data structures to the abstract data structures which were used in the user level file system in the debug phase.

4. The library calls the function that programmers wrote in the file system and that executes the file operation.

3.1.3 Protocol between the kernel and a user level file system

How granurality is best for a protocol between the kernel and a user level file system? Three protocols can be considered. One is the protocol in that the kernel does an upcall to a user level file system in the place where the kernel is designed so as to dispatch file operations according to file systems such as `VFS_MOUNT`. Another is the protocol in that libc traps system calls for files such as `mount` and executes

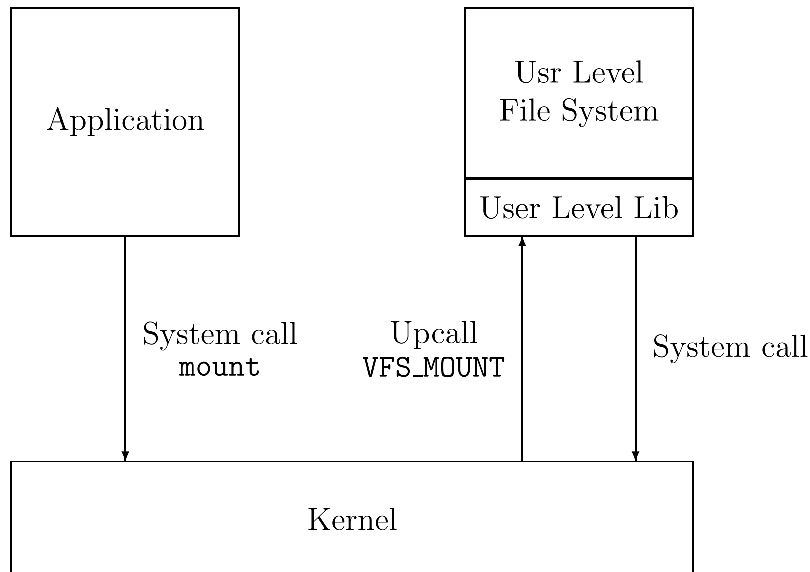

Figure 3.3: The protocol 1 between the kernel and a user level file system

them in the user space. The other is the protocol in that a system call for files does an upcall to a user level file system immediately after it enters the kernel.

The first protocol is illustrated like Figure 3.3. In this protocol, system call such as `mount` is executed in the kernel, and an upcall such as `VFS_MOUNT` is done to the user level file system from the system call. This protocol has two advantages. First, we can run the existing programs without any changes because we do not rewrite libc, where the real work of the system calls for files are done. If we rewrite libc, all programs which link libc statically must be relinked to new libc. Second, we can make use of the code which processes system calls for files in the kernel, so we can keep the library for a user level file system small.

However, this protocol also has two disadvantages. First, the context switch occurs more frequently as follows.

1. When an application program issues a system call for files, the context switch

to the kernel occurs.

2. The system call does some upcalls toward the user level file system to execute file operations such as VFS_MOUNT. Here, the context switch to the user level file system occurs.

Thus, one system call accompanies many context switches because one system call has more than one file operation in general. Second, we can not handle the data for a user level file system only at the user level because the part other than file operations in system calls is executed in the kernel, and the kernel also needs most of the data which we use in a user level file system. For that, most of the data must be managed in the kernel, and therefore a system call is needed every time we try to access the data. As a result the context switch increases, and the performance of a user level file system decreases remarkably.

The second protocol, on the other hand, is illustrated like Figure 3.4. In this protocol, libc communicates with a user level file system and processes system calls for files for the user level file system in the user space like Mach-US. This protocol has two advantages. First, it is unnecessary to call the kernel in system calls for a user level file system. This is because we rewrite libc so that system calls for a user level file system is executed in only libc only in the user space instead of calling the kernel. As a result, the performance for a user level file system is improved. Second, in relation to the above, the whole system becomes safer, (1) because a user level file system can execute most of the code in the user space, (2) because a user level file system can avoid executing the functions in the kernel code by means of system calls, and (3) because most of the data needed in a user level file system can be managed only in the user space.

However, this protocol also has two disadvantages. First, it is necessary to

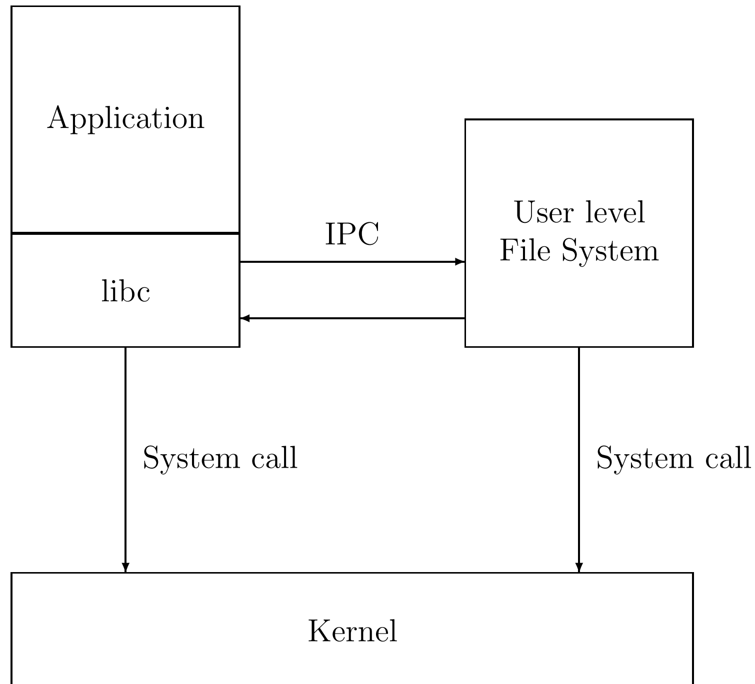

Figure 3.4: The protocol 2 between the kernel and a user level file system

relink the existing programs which link libc statically so that they use new libc. This change is very troublesome because almost all programs uses libc. Second, the amount of the code of libc increases because we must implement all system calls which use file operations such as `VFS_MOUNT` in libc. Fortunately, the size of an application program does not increase very much, since what is linked is only necessary system calls in libc.

The third protocol is illustrated like Figure 3.5. In this protocol, system call such as `mount` is passed to the user level file system as it is through the kernel. This protocol has two advantages. First, we need not change the existing programs like the first protocol, because system calls for files enter the kernel through existing libc, and does an upcall to a user level file system. Second, a user level file system can manage most of the data in the user space like the second protocol. Therefore,

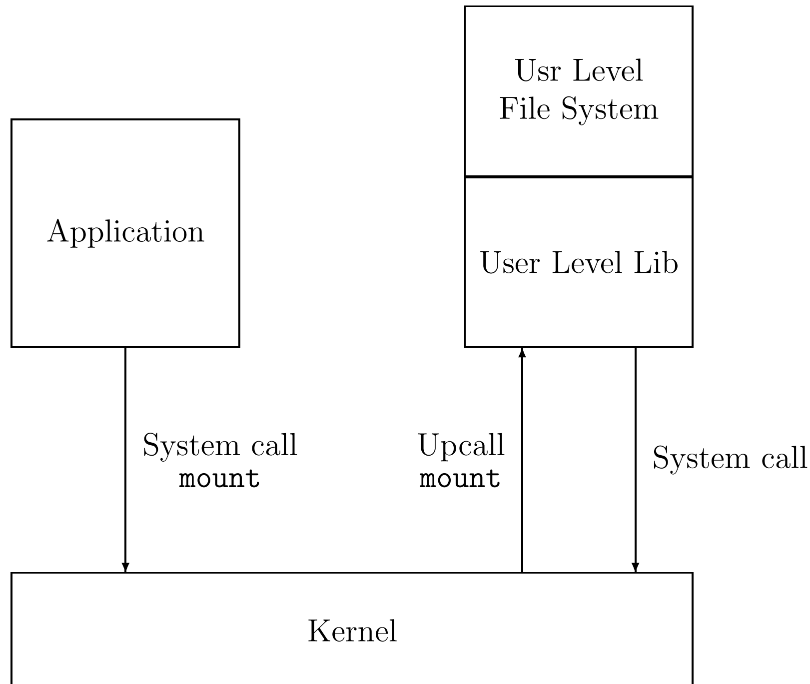

Figure 3.5: The protocol 3 between the kernel and a user level file system

the user level file system becomes safe, although the safety does not amount to that of the second protocol.

However, this protocol has two disadvantages. First, a user level file system is too big because we must implement all system calls which can use file operations and all functions which are called from these system calls. The number of such system calls and functions is more than 100. These all functions are linked to a user level file system. Second, the context switch increases more than that in the second protocol, although the context switch is less than that in the first protocol.

We have selected the first protocol for the following reasons.

- The number of the context switch, that is efficiency, is not very important because this protocol is used in the debug phase, which does not require the efficiency of a file system.

- We want to use existing programs without any changes.
- It takes much time to implement all system calls and related functions which can use file operations, as the user level library.
- If the library is too fat, the maintenance becomes cumbersome.

3.1.4 Passing data between the kernel and a user level file system

There are mainly two ways for passing data between the kernel and a user level file system: through the arguments of a system call and through the shared memory.

In the former way, we pass all contents that we need through the arguments of a system call. The number of a system call is only one. However, we must copy all data, including arrays, pointers and so on. Particularly, the handle of pointers is difficult because any user processes can not reference the contents of kernel pointers. Moreover, pointers are used as identifiers, so we also need the value of pointers. One way for the handle of pointers is to copy the contents referenced by the pointers, and add a value as an identifier. In this way, we may have to copy infinitely, when a pointer circulates.

For the above reason, we have chosen to use the shared memory. With the shared memory, we can use the pointer in communicating between the kernel and a user level file system. As a result, we can eliminate data copy. Furthermore, when we want to access the data on the shared memory, we can do so without any system calls. Therefore, this way is very efficient.

Of course, the way using the shared memory also has a disadvantage. We must map the shared memory when we enter the user level library, and unmap it when we leave the library. This is because the data on the shared memory may be destroyed if it is mapped when the code that programmers wrote in the user level

file system and that may be unsafe. To prevent this, excessive system calls, for map and unmap of the shared memory are required.

However, we want to handle data, particularly pointers, more easily, so we have chosen the way using the shared memory.

3.1.5 Data structure used in the communication

When the kernel and a user level file system can communicate each other, how structures should they use?

We have decided to use the data structures used in the kernel in the following reasons. If we use the data structure in the kernel for the communication, we must copy unnecessary data and it takes more time. Furthermore, the data consumes more large shared memory. However, this way has two advantages at least. One is that we can decrease the amount of the code in the kernel because it is possible to copy the data to the shared memory by means of `bcopy`. The other is that it is not necessary that we change the code of the kernel even if the data structures used in a the user level file system are changed.

On the other hand, the next is the reasons where we have not chosen the data structures of a user level file system for the communication. Although the amount of the copy of the data is minimum, the amount of the code of the kernel increases because we must copy the members of the data structures one by one. Moreover, we must change the code of the kernel if the data structures used in a user level file system is changed.

3.2 Extension of the kernel

We implement the pseudo file system in the kernel to do upcall toward the file system running at the user level, when the file operation for the file system is called.

3.2.1 Overview

In the kernel, the respective file operation is dispatched by the Virtual File System (VFS) [12], according to file systems. For example, when we mount UNIX File System (UFS), `VFS_MOUNT` is called from the kernel first, and then the `VFS_MOUNT` calls `ufs_mount`, which is the mount function for UFS. In `ufs_mount`, the real mount operation is done. So we must prepare entries such as `ufs_mount` for a user level file system. In fact, we make the pseudo file system in the kernel, which does upcalls toward a user level file system, because the user level file system exists at the user level and therefore the VFS function can not call the entry directly.

The function of the pseudo file system in the kernel does upcall toward the user level file system in the user space, instead of doing real file operation. Before that, we must put the arguments of the function into the global queue so that a user level file system, which is another process, can read that arguments.

Finally, after doing an upcall, the function of the pseudo file system sleeps until the real file operation finishes in a user level file system. If the file operation in a user level file system finishes, the process which sleeps in the kernel is waken up, and then continues to execute.

3.2.2 Implementation of upcall

We have chosen the way using a signal for upcall. However, there are two ways for upcall: the way using a socket and the way using a signal. Here, we discuss why we have chosen the way using a signal.

If we use a socket for upcall, we first create a socket in the kernel and in a user level file system, and then connect these two sockets at TCP. It is unnecessary that a user level file system does any excessive system calls other than `recv` when it catches one upcall, because we can pass the kind of file operation and arguments of the function to a user level file system at the same time when we do the upcall. Moreover, if the networks are fast enough, it is possible to run a user level file system even on another machine by Remote Procedure Call (RPC) [18] with a socket. In this case, we do not have to create both a client and a server for the network file system like Sun Network File System (NFS) [19, 6], so we can make the user level file system faster.

Using a socket is, however, difficult for some reasons, in fact. There are two difficulties. One is the fact that a socket is more complicated than a signal. The complexity of a socket causes the performance of upcall to be reduced. The other is that it is not realistic to run a user level file system on another machine. Now, network band width is 10Mbps at the Ethernet, or 100Mbps even at the Fast Ethernet. If upcalls are done through such slow networks every time when upcalls are required, it is impossible to improve the performance.

A signal, on the other hand, is very simple and fast. We only call `psignal`. However, it is necessary that a user level file system do one system call after it caught the signal from the kernel because a signal can not pass any data as arguments. In the case of either a socket or a signal, we must do one system call,

so this is not important issue for efficiency. Therefore, we have chosen using a signal.

3.2.3 Kernel protection

The kernel must be safe from the violation of a user level file system. In general, the kernel is protected from other user processes by means of the supervisor mode and the user mode in UNIX. However, the system call which accesses the kernel data is required by realizing file systems at the user level, and therefore the kernel is exposed to danger. For example, if we pass illegal arguments to the system call to change the data in the kernel, the kernel may crash or become unstable. It is difficult to protect the kernel from such a violation because we can hardly decide whether the arguments of the system call are wrong or not.

We can, however, protect the kernel from the crash when a user level file system crashes for segmentation fault in the middle of processing an upcall. First, when a user level file system crashes, the data in the kernel may be incorrect, so it is better to restore the data to stable states and return an error for the file operation that the kernel executes. Moreover, the kernel must not send any longer sequential upcalls to a user level file system, which has crashed and does not exist. This is because the processes which do upcalls stop in the kernel and that may make the kernel unstable. Second, some processes may exist on the directories mounted by a user level file system when it crashed. In this case, the vnode[9], that is virtual node for a file, for a current working directory is that for dead file system, and therefore the kernel crashes when someone tries to access the vnode. To prevent this, we must change the current working directory to the directory belonging to another file system.

3.2.4 System calls for a user level file system

We added eight new system calls for realizing the user level file system. These system calls are basically used only in the user level library for supporting a user level file system, although the programmers can also use them through the functions provided by the library.

First, we added two system calls for declaring the start and end of a user level file system: `ufstart` and `ufend`. `ufstart` initializes the shared memory between the kernel and a user level file system and records the process to which upcall should be done with a signal, that is the process of a user level file system. `ufend` cleans up the kernel so that the kernel can not use the data regarding the user level file system and is kept safe. These routines are called at the beginning and end of a user level file system only once, respectively.

Second, we added a system call for waking up the sleeping process in the kernel: `ufwakeup`. `ufwakeup` is called when an upcall finishes. The argument of this system call is the pointer to the arguments of the file operation gotten in `ufgetop`, that is explained below. If this system call is not called, the process sleeping in the kernel is blocked forever.

Third, we added three system calls for getting the information from the kernel as to a user level file system: `ufgetop`, `ufgetvar`, and `kernfunc`. `ufgetop` is called immediately after an upcall, and returns which file operation did upcall to the user level file system and the pointer to the arguments of the file operation, that is on the shared memory. `ufgetvar` is called when we want to get a global variable in the kernel such as the boot time of the kernel and the root vnode of root file system. `kernfunc` calls a function in the kernel, which we can not execute at the user level. This system call is very dangerous because the kernel may crash

if some arguments are wrong. To prevent the kernel from crashing, we check the arguments as strictly as possible.

Finally, we added two system calls for the management of the shared memory between the kernel and a user level file system: `shmalloc` and `shfree`. `shmalloc` allocates the shared memory and `shfree` frees it. These system calls originate from `malloc` and `free` in libc in NetBSD 1.2. However, we rewrote `malloc` with 4 byte alignment, because 64 bit data type such as `quad_t` is used in the kernel and therefore `shmalloc` requires 8 byte alignment. This shared memory is managed in the kernel unlike `malloc` and `free`, which are managed in the user space. The reason is that this shared memory is used by both the kernel and a user level file system, and moreover the management must be done atomically.

3.3 User level library

We implement a user level file system, that is the base of a user level file system. This library catches upcalls from the kernel and calls the function which programmers wrote for the user level file system. In short, this library mediates between the kernel and the code that programmers wrote. At this time, the kernel level data structures are translated the user level data structures with a higher level abstraction. In this section, we first describe the overview of this library, and then how abstract structures we provide.

3.3.1 Overview

The user level library is waiting for upcalls from the kernel and runs as follows by the upcall from the kernel.

1. When the library catches the upcall from the kernel, the process is forked

and issues a system call, `ufgetop`, to get the information which file operation the library should do and the arguments of the file operation.

2. The library creates the abstract data used in the user level file system from the arguments. This data is detached from both the kernel data and the shared memory, and is put in the user space.

3. The library calls the corresponding function that programmers wrote for the user level file system.

4. After the function finishes, the library wakes up the process which did the upcall and is sleeping in the kernel, using a system call, `ufwakeup`.

The function that programmers wrote may require that the kernel functions are executed. The kernel functions are realized as the emulation in the user level library or as the system call executing the kernel functions.

3.3.2 Multiplexing of the file system

We fork a user level file system immediately after it catches an upcall. This is for two reasons. One is because of increasing the throughput of a user level file system. If we do not fork the process of the user level file system at all, we can not process other upcalls during processing of one upcall. In brief, upcalls are processed sequentially. By means of forking the process every time one upcall comes, we can process more than one upcalls concurrently. However, if the overhead of the fork is bigger than the effect of concurrent processing of upcalls, the throughput is reduced, conversely.

In fact, we must necessarily fork the process for the other reason. The reason is because of the existence of the next upcall sequence.

1. The user level library catches the upcall and calls the function that programmers wrote.
2. The function requires to execute the kernel function, and the library issues the system call to do so.
3. In the system call, the upcall for other file operation is done.

At this time, the user level library has already blocked in the place where it issued the system call, so it can not catch any new upcalls. As a result, the user level file system freezes. To prevent this, we must fork a user level file system.

Although multiplexing should be realized with the kernel thread in point of the efficiency, NetBSD 1.2 does not support the kernel thread, so we have used the way to fork the process of the user level file system as the second best policy.

3.3.3 Higher-level abstraction

We provide a higher level abstraction for a user level file system. The abstraction is the simplification of the data structure and the increment of the granurality of the code.

For the simplification, we eliminate the unnecessary members from the kernel level data structures. The unnecessary members means what are not used in each individual file operation such as `VFS_MOUNT`. As a result, programmers becomes easy to understand these structure because the members that they need not see are hided. Furthermore, we simplify the parts that are complicated for efficiency in the kernel. For example, the structure of mbuf, which is mainly used as the network buffer in the kernel, is rather complicated because it is implemented so as to avoid copying data as much as possible. Therefore, it is troublesome to handle this structure as it is. So we provide the better interface with which we

need not operate complicated mbuf directly. Owing to this interface, we can also implement mbuf as a flat buffer in the user level library, and therefore decrease the bugs. These abstraction correspond to the encapsulation in object-oriented languages [5].

Second, we provide the functions which consist of a series of codes used frequently in many file systems, in the user level library. As far as programmers do not need specific routines, these functions help them to write file systems. Furthermore, we can also control the flow itself of each file operation in the library. In other words, programmers can write only different parts for each file system, which are called from the library.

3.4 Kernel level library

In this section, we explain the kernel level library that is used when the user level file system is embedded into the kernel in the release phase.

This library basically has three roles. The first role is to translate the kernel level data structures to the abstract data structures used in the user level file system, and vice versa. Because the data structure that we use in the user level file system is simplified, we must change it to fit in the kernel. To make this translation easy, we access the abstract data structure through the functions indirectly, not directly by means of the members.

The second role is to replace the functions realized in the user level library with what can run in the kernel. We emulate many functions in the kernel as the functions with the almost same fashion at the user level. In this cases, we simply can replace the functions at the user level with the corresponding functions at the kernel level by means of macros in C. Otherwise, we must prepare new functions

in this library.

The third role is to enable the kernel to call the user level file system. To make the kernel to recognize this new file system, this library provides dispatch tables for file operations, and registers the file system in the kernel.

Chapter 4

Experiment

This chapter mentions the file system that we implemented on our system and the experiment using this file system, and discusses the result.

The aim of this experiment is that we verify that the file system implemented with our system is nearly as efficient as the file system implemented in the kernel from the beginning, after the user level file system implemented on our system is embedded into the kernel. Even if it is easy to implement a file system, the file system is unuseful if it is too inefficient.

First, we implement the simple memory file system (SMFS), that is RAM disk in the kernel. This file system possesses only minimum facilities such as create, remove, read, and write; it does not possess facilities such as mkdir, symlink and rename. Moreover, this file system does not cache directories and so on, and therefore our experiment does not influence any caches. We call this file system an original SMFS. Second, we implement SMFS on our system as a user level file system. We call this file system a user level SMFS. Finally, we embed the user level SMFS into the kernel. We call this file system a kernel level SMFS.

Next, we measure the performance of these three file systems. The environment of this experiment is as follows:

File Operation	Number of Upcall	File Operation	Number of Upcall
lock	260	close	2
unlock	260	getattr	2
strategy	258	inactive	2
read	128	fsync	2
write	128	reclaim	2
lookup	2	access	1
open	2	create	1

Table 4.1: The number of upcalls to the user level SMFS in this experiment.

- SPARCstation 5 (MicroSPARC2 85MHz, Memory 32M)
- NetBSD 1.2 for sparc

The reason where we have chosen SPARCstation 5 is that it spreads widely although it is not the fastest machine.

There are various ways to measure the performance of a file system. In those ways, we consider that the most important operations which a file system does is read and write, and then we measure the performance of read and write. To measure the performance, we use the following simple procedure.

1. Open file “A” and “B”
2. Read data of 64K byte from file “A”
3. Write the data to file “B”
4. Close file “A” and “B”

File System	Time(ms)	Ratio
SMFS	49	1
Embedded SMFS	55	1.12
User Level SMFS	23,830	486

Table 4.2: The time spent on copy of 64k byte and the ratio to the original SMFS.

Table 4.1 shows the number of upcalls to the user level SMFS in this procedure. **Strategy** operation does real work of read and write. The block size in SMFS is 512 byte, and therefore **read** and **write** operations are done 128[1] times, respectively. One set of **read** or **write** operation is executed as follows:

1. Lock
2. Read or write
3. Strategy
4. Unlock

The other operations are called for open or close operation.

Table 4.2 shows the result of above procedure. From this result, we can see that the performance of the kernel level SMFS implemented with our system is almost same as the original SMFS implemented in the kernel from the beginning. Although the user level SMFS is very inefficient, this results from the pursuit of the ease of implementation of file systems, so this inefficiency is negligible.

Because this file system is very simple, the overhead of the abstraction is also small. So the difference between the file system implemented with the abstract

[1] 65,536/512

data structures and that with the kernel level data structures is not large. If the objective file system is more complicated and the effect of the abstraction is exhibited fully, the difference will become large. However, because it is considered that the difference is not too large, even if so, it was shown that our approach is appropriate.

Chapter 5

Conclusion

In this chapter, we summarize our study and discuss our future work.

5.1 Summary

This thesis proposed the framework with that programmers can first implement a file system at the user level and then embed it into the kernel, called two-phase implementation.

The conventional kernels lack the balance of efficiency and ease of implementation. The monolithic kernel pursues only efficiency by implementing the file systems in the kernel and running them in the kernel space. Conversely, the microkernel pursues only ease of implementation by implementing the file systems as the servers in the user space and running them in the user space.

To solve these unbalanced systems, we separated the debug phase and the release phase. The debug phase realizes ease of implementation, whereas the release phase realizes efficiency. Therefore, two-phase implementation enables file systems to satisfy these two antithetic properties. In other word, our system is balanced.

Finally, to show the feasibility of our idea, we implemented the simple local

file system on top of our system, and then measured the performance after we embedded the file system into the kernel. From this experiment, we concluded that the file system implemented with our system has the almost same performance as the file system implemented in the kernel from the beginning.

5.2 Future work

Now, our system makes use of only a local file system. This is the first step of our work. The next step is to apply this system to a network file system. As the development of mobile computing, the network file systems, such as Andrew File System (AFS) [7] and Coda file system [15], become more important gradually. If such network file systems are implemented with our system, their studies must progress faster.

Furthermore, to apply this system to various kinds of file systems, we must provide more abstract interface. For that, it is important to introduce more object-oriented concepts.

Our final goal is to apply this framework to not only file systems, but also the whole system. This framework allows us to change the policies of various facilities such as schedulers, virtual memory, thread, and so on easily, and to keep the efficiency of the final release as high as the monolithic kernel.

References

[1] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, “Mach: a new kernel foundation for UNIX development,” *Proceedings of the USENIX 1986 Summer Conference (Atlanta)*, pp. 93–112, Jun. 1986.

[2] Barrera, III, J. S., “A Fast Mach Network IPC Implementation,” in *Proceedings of the USENIX Mach Symposium: November 20–22, 1991, Monterey, California, USA* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 1–12, USENIX, Nov. 1991.

[3] Bershad, B., S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers, “Extensibility, Safety and Performance in the SPIN Operating System,” in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267–284, Copper Mountain, CO., Dec. 1995.

[4] Bershad, B. N., C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. aw Pardyak, S. Savage, and E. Gün Sirer, “SPIN – an Extensible Microkernel for Application-Specific Operating System Services,” tech. rep., Department of Compuer Science and Engineering, Univ. of Washingt on, February 1994.

[5] Ewing, J. J., “An Object-Oriented Operating System Interface,” in *Proceedings OOPSLA ’86, ACM SIGPLAN Notices*, pp. 46–56, Nov. 1986. Published as Proceedings OOPSLA ’86, ACM SIGPLAN Notices, volume 21, number 11.

[6] Gould, E., “The Network File System Implemented on 4.3BSD,” in *USENIX Conference Proceedings*, (Atlanta, GA), pp. 294–298, USENIX, Summer 1986.

[7] Howard, J. H., “On Overview of the Andrew File System,” in *USENIX Conference Proceedings*, pp. 23–6, USENIX, Winter 1988.

[8] Julin, D., J. Chew, J. M. Stevenson, P. Guedes, R. Neves, and P. Roy, “Generalized Emulation Services for Mach 3.0 Overview, Experiences and Current Status,” in *The Second USENIX Mach Symposium Conference Proceedings*, USENIX, Nov. 1991.

[9] Kleiman, S. R., “Vnodes: An Architecture for Multiple File System Types in Sun UNIX,” in *USENIX Conference Proceedings*, (Atlanta, GA), pp. 238–247, USENIX, Summer 1986.

[10] Leffler, S. J., M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*. Computer Science, Addison Wesley, may 1989. ISBN 0-201-06196-1.

[11] McKusick, M. K., K. Bostic., M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[12] McKusick, M. K., “The Virtual Filesystem Interface in 4.4BSD,” *Computing Systems*, vol. 8, pp. 3–25, Winter 1995.

[13] Mullender, S., *DISTRIBUTED SYSTEMS*. Addison Wesley, second ed., 1993.

[14] Nelson, G., *System Programming with Modula-3.* Innovative Technology, Prentice Hall, 1991. ISBN 0-13-590464-1.

[15] Satyanarayanan, M., J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, “Coda: A highly available file system for a distributed workstation environment,” *IEEECOMP.*, vol. 39, no. 4, pp. 447–59, Apr.

[16] Silberschatz, A., J. L. Peterson, and P. B. Galvin, *Operating System Concepts.* Addison Wesley, third ed., 1991.

[17] Small, C. H., “UNIX Operating Systems,” *EDN*, p. 102, May 17 1984.

[18] Sun Microsystems, Inc., Mountain View, CA, *Remote Procedure Call Protocol Specification*, Feb. 1986.

[19] Sun Microsystems, Inc., “NFS: Network File System Protocol Specification,” RFC 1094, Network Information Center, SRI International, Mar. 1989.

Appendix A

A program for our experiment

The following program is the program that we used for measuring the performance of three file system in Chapter 4.

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

#define SIZE 65536

char buf[SIZE];

main()
{
  int n;
  FILE *fin, *fout;
  struct timeval tv1, tv2;
  long elapse;

  gettimeofday(&tv1, NULL);

  if ((fin = fopen("a", "r")) == NULL) {
    perror("fopen: a");
    exit(1);
  }
  if ((fout = fopen("b", "w")) == NULL) {
    perror("fopen: b");
    exit(1);
  }

  if ((n = fread(buf, 1, SIZE, fin)) != SIZE) {
    perror("fread: a");
    exit(1);
  }
  if ((n = fwrite(buf, 1, SIZE, fout)) != SIZE) {
    perror("fwrite: b");
    exit(1);
  }
```

```
  fclose(fin);
  fclose(fout);

  gettimeofday(&tv2, NULL);

  elapse = (tv2.tv_sec-tv1.tv_sec)*1000000 + tv2.tv_usec-tv1.tv_usec;

  printf("elapse: %ld us\n", elapse);
}
```